Nim's Memory Management 1.3.5

Andreas Rumpf

August 21, 2020

"The road to hell is paved with good intentions."

1 Introduction

This document describes how the multi-paradigm memory management strategies work. How to tune the garbage collectors for your needs, like (soft) realtime systems, and how the memory management strategies that are not garbage collectors work.

2 Multi-paradigm Memory Management Strategies

To choose the memory management strategy use the `-gc:` switch.

- `-gc:refc`. This is the default GC. It's a deferred reference counting based garbage collector with a simple Mark&Sweep backup GC in order to collect cycles. Heaps are thread local.
- `-gc:markAndSweep`. Simple Mark-And-Sweep based garbage collector. Heaps are thread local.
- `-gc:boehm`. Boehm based garbage collector, it offers a shared heap.
- `-gc:go`. Go's garbage collector, useful for interoperability with Go. Offers a shared heap.
- `-gc:arc`. Plain reference counting with move semantic optimizations, offers a shared heap. It offers deterministic performance for hard realtime systems. Reference cycles cause memory leaks, beware.
- `-gc:orc`. Same as `-gc:arc` but adds a cycle collector based on "trial deletion". Unfortunately that makes its performance profile hard to reason about so it is less useful for hard realtime systems.
- `-gc:none`. No memory management strategy nor garbage collector. Allocated memory is simply never freed. You should use `-gc:arc` instead.

JavaScript's garbage collector is used for the JavaScript and NodeJS compilation targets. The NimScript target uses the memory management strategy built into the Nim compiler.

Memory Management	**Heap**	**Reference Cycles**	**Stop-The-World**	**Command line switch**
RefC	Local	Cycle Collector	No	`-gc:refc`
Mark & Sweep	Local	Cycle Collector	No	`-gc:markAndSweep`
ARC	Shared	Leak	No	`-gc:arc`
ORC	Shared	Cycle Collector	No	`-gc:orc`
Boehm	Shared	Cycle Collector	Yes	`-gc:boehm`
Go	Shared	Cycle Collector	Yes	`-gc:go`
None	Manual	Manual	Manual	`-gc:none`

3 Tweaking the refc GC

3.1 Cycle collector

The cycle collector can be en-/disabled independently from the other parts of the garbage collector with `GC_enableMarkAndSweep` and `GC_disableMarkAndSweep`.

3.2 Soft realtime support

To enable realtime support, the symbol useRealtimeGC needs to be defined via `-define:useRealtimeGC` (you can put this into your config file as well). With this switch the garbage collector supports the following operations:

```
proc GC_setMaxPause*(maxPauseInUs: int)
proc GC_step*(us: int, strongAdvice = false, stackSize = -1)
```

The unit of the parameters `maxPauseInUs` and `us` is microseconds.
These two procs are the two modus operandi of the realtime garbage collector:
(1) GC_SetMaxPause Mode

You can call `GC_SetMaxPause` at program startup and then each triggered garbage collector run tries to not take longer than `maxPause` time. However, it is possible (and common) that the work is nevertheless not evenly distributed as each call to `new` can trigger the garbage collector and thus take `maxPause` time.

(2) GC_step Mode

This allows the garbage collector to perform some work for up to `us` time. This is useful to call in a main loop to ensure the garbage collector can do its work. To bind all garbage collector activity to a `GC_step` call, deactivate the garbage collector with `GC_disable` at program startup. If `strongAdvice` is set to `true`, then the garbage collector will be forced to perform collection cycle. Otherwise, the garbage collector may decide not to do anything, if there is not much garbage to collect. You may also specify the current stack size via `stackSize` parameter. It can improve performance, when you know that there are no unique Nim references below certain point on the stack. Make sure the size you specify is greater than the potential worst case size.

These procs provide a "best effort" realtime guarantee; in particular the cycle collector is not aware of deadlines. Deactivate it to get more predictable realtime behaviour. Tests show that a 1ms max pause time will be met in almost all cases on modern CPUs (with the cycle collector disabled).

3.3 Time measurement with garbage collectors

The garbage collectors's way of measuring time uses (see `lib/system/timers.nim` for the implementation):

1. `QueryPerformanceCounter` and `QueryPerformanceFrequency` on Windows.
2. `mach_absolute_time` on Mac OS X.
3. `gettimeofday` on Posix systems.

As such it supports a resolution of nanoseconds internally; however the API uses microseconds for convenience.

Define the symbol `reportMissedDeadlines` to make the garbage collector output whenever it missed a deadline. The reporting will be enhanced and supported by the API in later versions of the collector.

3.4 Tweaking the garbage collector

The collector checks whether there is still time left for its work after every `workPackage`'th iteration. This is currently set to 100 which means that up to 100 objects are traversed and freed before it checks again. Thus `workPackage` affects the timing granularity and may need to be tweaked in highly specialized environments or for older hardware.

4 Keeping track of memory

If you need to pass around memory allocated by Nim to C, you can use the procs `GC_ref` and `GC_unref` to mark objects as referenced to avoid them being freed by the garbage collector. Other useful procs from system you can use to keep track of memory are:

- `getTotalMem()` Returns the amount of total memory managed by the garbage collector.
- `getOccupiedMem()` Bytes reserved by the garbage collector and used by objects.
- `getFreeMem()` Bytes reserved by the garbage collector and not in use.
- `GC_getStatistics()` Garbage collector statistics as a human-readable string.

These numbers are usually only for the running thread, not for the whole heap, with the exception of `-gc:boehm` and `-gc:go`.

In addition to `GC_ref` and `GC_unref` you can avoid the garbage collector by manually allocating memory with procs like `alloc`, `alloc0`, `allocShared`, `allocShared0` or `allocCStringArray`. The garbage collector won't try to free them, you need to call their respective *dealloc* pairs (`dealloc`, `deallocShared`, `deallocCStringArray`, etc) when you are done with them or they will leak.

5 Heap dump

The heap dump feature is still in its infancy, but it already proved useful for us, so it might be useful for you. To get a heap dump, compile with `-d:nimTypeNames` and call `dumpNumberOfInstances` at a strategic place in your program. This produces a list of used types in your program and for every type the total amount of object instances for this type as well as the total amount of bytes these instances take up. This list is currently unsorted! You need to use external shell script hacking to sort it.

The numbers count the number of objects in all garbage collector heaps, they refer to all running threads, not only to the current thread. (The current thread would be the thread that calls `dumpNumberOfInstances`.) This might change in later versions.